Electronic Notes in Theoretical Computer Science 4 (1996)

A reflective extension of ELAN

Hélène Kirchner, Pierre-Etienne Moreau

INRIA Lorraine & CRIN-CNRS
615, rue du Jardin Botanique, BP 101
54602 Villers-lès-Nancy Cedex, France
http://www.loria.fr/equipe/protheo.html
email: Helene.Kirchner@loria.fr,moreau@loria.fr

Abstract

The expressivity of rewriting logic as meta-logic has been already convincingly illustrated. The goal of this paper is to explore the reflective capabilities of ELAN, a language based on the concepts of computational systems and rewriting logic. We define a universal theory for the class of ELAN programs and the representation function associated to this universal theory. Then we detail the effective transformations to implement and propose the definition of two built-in modules that provide the last step to get the reflective capabilities we want for the ELAN system.

1 Introduction

The expressivity of rewriting logic as a meta-logic able to encode various object logics, has been already convincingly illustrated in [Mes92,MOM93,KKV95a]. Reflective properties of rewriting logic are studied for instance in [CM96], in which metalogical axioms for reflective logics and declarative languages in general are proposed. The two notions of *universal theory* for a class of representable theories and of *representation function* are defined. In this approach the logic of choice is a parameter and the formalism is illustrated by the specific case of rewriting logic.

Our interest on reflection comes from the development of a declarative rewriting logic language called ELAN and the goal of this paper is to explore the reflective capabilities of ELAN.

ELAN is an environment dedicated to prototype, experiment and study the combination of different deduction systems for constraint solving, theorem proving and logic programming paradigms. The language is based on the concept of computational systems [KKV95a,Vit94,BKK+96b] given by a signature providing the syntax, a set of conditional rewrite rules describing the deduction mechanism, and a strategy to guide application of rewrite rules. Formally, this is a rewrite theory in rewriting logic [Mes92], [MOM93], together with a notion of strategy to select relevant computations. Strategy definitions in the currently distributed version of ELAN are based on a

small language combining labels of rewrite rules, a concatenation operator, an iterator `iterate`, and two selecting operators `dont know choose` and `dont care choose`, corresponding to non-deterministic and deterministic choices of strategies. Each ELAN module defines its own signature, labelled rewrite rules and strategies. ELAN is implemented in `C++`.

Four practical problems presented below motivate the design of a reflexive extension of ELAN.

Preprocessor: ELAN uses a pre-processing phase that allows describing the logic to be encoded in a flexible way. The pre-processor performs textual replacements and sometimes needs rewriting steps. So there are several interactions between the pre-processor and the abstract rewriting machine. In a reflective extension, the program written in the pre-processor syntax can be viewed as an ELAN term in an extended syntax, and can be transformed naturally by rewriting, using rewrite rules and strategies describing the pre-processing phase. Beyond the fact that this provides a unified semantics for the pre-processor and the interpretor, this also makes compilation of the system easier.

Strategy language: In automated deduction it is now a common approach to use a metalanguage to write strategies and tactics, specifying how object logic inference rules are composed to build proofs. We have developed in [BKK96a] a powerful strategy language for ELAN that is reflective in the sense that it is defined in rewriting logic. In order to implement it as an extension of the actual system, a systematic enrichment of the signatures, new rules and strategies must be added for describing strategy evaluation. In the first prototype of the strategy language, those systematic extensions are automatically done by ELAN, but this necessitated to modify the ELAN native code. A reflective extension of ELAN would allow prototyping the abstract strategy interpreter without any modification in the native code, as an extension of the computational systems.

Completion: In equational logic and equational programming languages, the Knuth-Bendix completion algorithm (or its variants) provides a way, when it succeeds, to transform an equational theory into a terminating and confluent set of rewrite rules with the same deductive power. Completion procedures, as many computational processes, can be formulated as instances of a schema that consists of applying rewrite rules on formulas with some strategy, until getting specific normal forms. In this sense they can be understood as computational systems. We have described in [KM95] completion algorithms in ELAN, in which rules to be completed are represented by terms, and the mechanism of simplification (ie. rewriting steps) is described in ELAN. In a reflective extension, we could transform terms, representing rules, into rewrite rules of the rewriting machine and use them to simplify the set of rules (represented by a set of terms) with the efficient ELAN rewriting mechanism.

Memorisation: Another possible use in ELAN of reflective capability would be for efficiency. Let $\mathcal{R}$ be a rewrite system, t a term and t' a normal form

of t w.r.t. $\mathcal{R}$. Let us assume that the normal form t' has to be computed several times; a reflective extension should permit adding the rewrite rule $t \Rightarrow t'$ and would transform $\mathcal{R}$ into a more efficient program.

In order to deal with program modification or extension, we can identify several problems to solve: representing (coding) programs by terms, simulating their execution, translating forth and back from programs to their representations, justifying the equivalence of deduction in both worlds of programs and their term representations.

In Section 2, we recall and adapt rewriting logic and universal theory to those rewrite theories which are actually used in ELAN with in particular conditions and local affectations in rewrite rules. In Section 3, we define an ELAN program that implements a universal theory for the class of ELAN programs and define the representation function associated to this universal theory. Then in Section 4, we detail the effective transformations to implement in order to make the system reflective. We propose the definition of two built-in modules that should be implemented in order to get the reflective capabilities we want for the ELAN system. Section 5 mentions some related work.

2 General setting

This section briefly presents the main concepts of general logic [Mes89] and rewriting logic [Mes92]. We slightly generalise the syntax and proof theory of conditional rewriting logic by introducing rules with local affectations. The notion of universal theory proposed by [CM96] is used to formalise metalogical axioms for reflective logics and languages.

2.1 Rewriting Logic

The definitions below are given in the unsorted case. The many-sorted and order-sorted cases can be handled in a similar, although more technical, manner. Our definitions are consistent with [DJ90,JK91] to which the reader is referred for more detailed considerations on universal algebra, term rewriting systems and matching.

We consider a set $\mathcal{F}$ of ranked function symbols, where $\mathcal{F}_n$ is the subset of functions of arity n, a set $\mathcal{X}$ of variables and the set of first-order terms $\mathcal{T}(\mathcal{F}, \mathcal{X})$ built on $\mathcal{F}$ and $\mathcal{X}$.

A *substitution* is an assignment from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$, written $\sigma = \{y_1 \mapsto t_1, \ldots, y_k \mapsto t_k\}$. It uniquely extends to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We also use the notation $\{\overline{w} \mapsto \overline{x}\}(t)$ to express the simultaneous substitution of w_i for x_i in t. Letters $\sigma, \mu, \gamma, \phi, \ldots$ denote substitutions, and $\circ$ denotes their composition.

A $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equality is a pair of terms $\{t, t'\}$ in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, denoted as usual $(t = t')$. For any set of equalities E, $\mathcal{T}(\mathcal{F}, \mathcal{X})/E$ denotes the free quotient algebra of terms modulo E. The equivalence class of a term t modulo E is denoted $\langle t \rangle_E$ or just $\langle t \rangle$. For details and general results on calculus modulo

equational axioms, the reader is invited to consult for example [JK86]. To simplify notation, we denote a sequence of objects $(a_1, \ldots, a_n)$ by $\overline{a}$ or $\overline{a}^n$.

Syntax.

The syntax needed for defining a logic is provided by a signature which allows building sentences. In rewriting logic, a signature consists of a 3-tuple $\Sigma = (\mathcal{L}, \mathcal{F}, E)$, where $\mathcal{L}$ and $\mathcal{F}$ are sets of ranked function symbols and E is a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities. Sentences built on a given signature are defined as sequents of the form $\pi \quad : \quad \langle t \rangle \rightarrow \langle t' \rangle$ where $t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and π is the proof term representing the proof that allows to derive t' from t. So in rewriting logic, proofs are first-order objects identified with proof terms. In order to compose proofs, we introduce the infix binary operator "$\_;\_$" on proof terms. In order to record subproofs corresponding to conditions or local affectations, we introduce the operator "$\_\{\_\}$" whose second argument is a list of subproofs of the first argument. Therefore a proof term is by definition a term built on equivalence classes of $\mathcal{T}(\mathcal{F}, \mathcal{X})/E$, function symbols in $\mathcal{F}$, label symbols in $\mathcal{L}$, the composition operator "$\_;\_$", the subproof operator "$\_\{\_\}$". In other words, the set of proof terms is the term algebra $\mathcal{PT} = \mathcal{T}(\mathcal{L} \cup \{\_;\_\ ,\ \_\{\_\}\} \cup \mathcal{F} \cup \mathcal{T}(\mathcal{F}, \mathcal{X})/E)$. Lists of proof terms are built from the empty list "$\Box_{pt}$" and the concatenation operator "$\_ \cdot \_$".

Since we need to be generic, we consider $\mathcal{S}ynt$ a class of pairs (Σ, sen) consisting of a signature Σ together with a mapping sen associating to Σ the set of all legal sentences built on this signature.

Entailment systems.

For a given class of syntax $\mathcal{S}ynt$ and (Σ, sen) in $\mathcal{S}ynt$, a theory T presented by a set of axioms Φ is the pair $T = (\Sigma, \Phi)$ where $\Phi \subseteq sen(\Sigma)$. Given a signature Σ, an entailment system is an abstract description of the provability relation of a sentence ϕ starting from a given set of sentences (also called axioms) Φ and using logical rules.

In rewriting logic, in order to build the entailment system, the notion of rewrite theory is introduced and an appropriate deduction system allows to inductively define the entailment relation.

A labelled rewrite theory is a 5-tuple $\mathcal{R} = (\mathcal{X}, \mathcal{F}, E, \mathcal{L}, R)$ where $\mathcal{X}$ is a given countably infinite set of variables, $\mathcal{L}$ and $\mathcal{F}$ are sets of ranked function symbols, E a set of $\mathcal{T}(\mathcal{F}, \mathcal{X})$-equalities, and R is a set of rewrite rules denoted

$$\begin{aligned} \ell \quad : \quad \langle u \rangle \rightarrow \langle u' \rangle \quad & \textbf{if } \langle v_1 \rangle \rightarrow \langle v_1' \rangle \wedge \ldots \wedge \langle v_j \rangle \rightarrow \langle v_j' \rangle \\ & \textbf{where } \sigma = \{y_1 \mapsto \langle a_1 \rangle, \ldots, y_k \mapsto \langle a_k \rangle\} \end{aligned}$$

The part **if** $\langle v_1 \rangle \rightarrow \langle v_1' \rangle \wedge \ldots \wedge \langle v_j \rangle \rightarrow \langle v_j' \rangle$ is called the *condition* of the rule and **where** $\sigma = \{y_1 \mapsto \langle a_1 \rangle, \ldots, y_k \mapsto \langle a_k \rangle\}$ the *local affectation* of the rule. A rewrite rule may involve variables in its right-hand side that do not occur in its left-hand side, provided they are instantiated in the **where** part. To

indicate that $\{x_1, \ldots, x_n\}$ is the set of variables occuring in either u or u', we write $u(x_1, \ldots, x_n) \to u'(x_1, \ldots, x_n)$.

A given labelled rewrite theory $\mathcal{R}$ entails the sequent $\pi \quad : \quad \langle t \rangle \to \langle t' \rangle$, which is denoted $\mathcal{R} \vdash \pi \quad : \quad \langle t \rangle \to \langle t' \rangle$, if $\pi \quad : \quad \langle t \rangle \to \langle t' \rangle$ is obtained by some finite application of the deduction rules in FIG. 1.

Reflexivity

$$t \quad : \quad \langle t \rangle \to \langle t \rangle$$

Congruence

$$\frac{\pi_i \quad : \quad \langle t_i \rangle \to \langle t'_i \rangle, \ i = 1 \ldots n}{f(\pi_1, \ldots, \pi_n) \quad : \quad \langle f(t_1, \ldots, t_n) \rangle \to \langle f(t'_1, \ldots, t'_n) \rangle}$$

Replacement. For each rewrite rule

$[l] \quad \langle u(\overline{x}) \rangle \to \langle u'(\overline{x} \cup \overline{y}) \rangle$
if $\langle v_1(\overline{x} \cup \overline{y}) \rangle \to \langle v'_1(\overline{x} \cup \overline{y}) \rangle \wedge \ldots \wedge \langle v_j(\overline{x} \cup \overline{y}) \rangle \to \langle v'_j(\overline{x} \cup \overline{y}) \rangle$
where $\sigma = \{y_1 \mapsto a_1(\overline{x}), \ldots, y_k \mapsto a_k(\overline{x} \cup \overline{y}^{k-1})\}$

$$\frac{\begin{array}{rcl} \alpha_i & : & \langle w_i \rangle \to \langle w'_i \rangle, \ i = 1 \ldots n \\ \beta_1 & : & \langle a_1(\{\overline{x} \mapsto \overline{w}\}) \rangle \to \langle a'_1(\{\overline{x} \mapsto \overline{w}\}) \rangle \\ & & \vdots \\ \beta_k & : & \langle a_k(\{\overline{x} \cup \overline{y}^{k-1} \mapsto \overline{w} \cup \overline{a}^{k-1}\}) \rangle \to \langle a'_k(\{\overline{x} \cup \overline{y}^{k-1} \mapsto \overline{w} \cup \overline{a}^{k-1}\}) \rangle \\ \gamma_1 & : & \langle v_1(\{\overline{x} \cup \overline{y} \mapsto \overline{w} \cup \overline{a}\}) \rangle \to \langle v'_1(\{\overline{x} \cup \overline{y} \mapsto \overline{w} \cup \overline{a}\}) \rangle \\ & & \vdots \\ \gamma_j & : & \langle v_j(\{\overline{x} \cup \overline{y} \mapsto \overline{w} \cup \overline{a}\}) \rangle \to \langle v'_j(\{\overline{x} \cup \overline{y} \mapsto \overline{w} \cup \overline{a}\}) \rangle \end{array}}{\ell(\overline{\alpha}^n)\{\overline{\beta}^k.\overline{\gamma}^j\} \quad : \quad \langle u(\{\overline{x} \mapsto \overline{w}\}) \rangle \to \langle u'(\{\overline{x} \cup \overline{y} \mapsto \overline{w}' \cup \overline{a}'\}) \rangle}$$

Transitivity

$$\frac{\pi_1 \quad : \quad \langle t_1 \rangle \to \langle t_2 \rangle \quad \pi_2 \quad : \quad \langle t_2 \rangle \to \langle t_3 \rangle}{(\pi_1; \pi_2) \quad : \quad \langle t_1 \rangle \to \langle t_3 \rangle}$$

Fig. 1. Deduction rules for Rewriting Logic

The **Replacement** rule looks more complex than its version in [Mes92] for instance, due to the additional introduction of local affectations. Its hypotheses can be split into three parts: the first one (with proof terms α_i) for the deductions in the substitution part, the second one (with proof terms β_i) for the deductions in the local affectation part, and the third one (with proof terms γ_i) for the deductions in the condition part. But this is actually just a syntactical facility that could be handled in the same way as conditions in [Mes92]. Especially the proof term $\ell(\overline{\alpha}^n)\{\overline{\beta}^k.\overline{\gamma}^j\}$ is actually another notation for a proof term $\ell(\overline{\alpha}^n, \overline{\beta}^k, \overline{\gamma}^j)$ in the notation of [Mes92]. We prefer the more structured first notation that seems more readable to us. This corre-

spondence between the two notations allows us to directly reuse the results of [Mes92].

An equivalence on proof terms is defined by E and a set $E_{PT(R)}$ of equational axioms described in FIG. 2. This equivalence relation is important to relate different derivations with the same result, but different proofs.

$\forall \pi_1, \pi_2, \pi_3 \in PT \quad \pi_1;(\pi_2;\pi_3) = (\pi_1;\pi_2);\pi_3$	Associativity
$\forall \pi : \langle t \rangle \to \langle t' \rangle, \quad \pi;\langle t' \rangle = \pi, \text{ and } \langle t \rangle;\pi = \pi$	Local Identities
For all $f \in \mathcal{F}_n, n \in Nat, \quad \forall \pi_1, \ldots, \pi_n, \pi'_1, \ldots, \pi'_n$: $f(\pi_1;\pi'_1, \ldots, \pi_n;\pi'_n) = f(\pi_1, \ldots, \pi_n);f(\pi'_1, \ldots, \pi'_n)$	Independence
For all $f \in \mathcal{F}_n, n \in Nat$: $f(\langle t_1 \rangle, \ldots, \langle t_n \rangle) = \langle f(t_1, \ldots, t_n) \rangle$	Preservation of E
$\forall \ell : g \to d \in R, \forall \pi_1 : \langle t_1 \rangle \to \langle t'_1 \rangle, \ldots, \pi_n : \langle t_n \rangle \to \langle t'_n \rangle$ $\ell(\pi_1, \ldots, \pi_n) = \ell(\langle t_1 \rangle, \ldots, \langle t_n \rangle);d(\pi_1, \ldots, \pi_n)$ and $\ell(\pi_1, \ldots, \pi_n) = g(\pi_1, \ldots, \pi_n);\ell(\langle t'_1 \rangle, \ldots, \langle t'_n \rangle)$	Parallel Move Lemma

Fig. 2. $E_{PT(R)}$: Equivalence of proof terms

2.2 Universal theory for Rewriting Logic

A reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way. Two metatheoretic notions that can be reflected are theories and the entailment relation $\vdash$. This leads to the notion of universal theory, relative to a class C of *representable* theories proposed by [CM96].

For a given theory T, let $proofs(T)$ denote the set of all the proofs of theorems of the theory T.

Definition 2.1 Given an entailment system $\mathcal{E}$ and a class of theories C, a theory U is C-universal if there is a function, called *representation function*,

$$\overline{(\_ \vdash \_ : \_)} : \cup_{T \in C}\{T\} \times proofs(T) \times sen(T) \longrightarrow sen(U)$$

such that for each $T \in C, p \in proofs(T), \varphi \in sen(T)$, $T \vdash p : \varphi \iff U \vdash \overline{T \vdash p : \varphi}$.

If, in addition, $U \in C$, then the entailment system $\mathcal{E}$ is called *C-reflective*.

Note that, if U is itself representable, we have a "reflective tower":

$$T \vdash p : \varphi \iff U \vdash q : \overline{T \vdash p : \varphi} \iff U \vdash q' : \overline{U \vdash q : \overline{T \vdash p : \varphi}}$$

In the appendix of [CM96], the authors give an example of a universal theory U for rewriting logic. Then, a representation function for U is defined with an overloaded symbol $\overline{(\_)}$, described recursively from the representation

of theories, rewrite rules, terms, etc. In what follows we concentrate on the class C of finitely presentable conditional rewrite theories with local affectations that are the basis for ELAN. The problem is to find a theory U and a representation function making U a C-universal theory.

3 Universal Theory and ELAN

The notions of universal theory and representation function introduced in the previous section can be explained in terms of meta-level objects (such as signatures or sets of labelled rewrite rules and strategies) and base-level objects that are only terms. Using the terminology of [KSO95], we can call *meta-transformation* the representation function, that transforms meta-level objects into base-level objects, and *base-transformation* the converse transformation. In order to distinguish syntactically these two levels, different signatures are introduced. An ELAN program defines in a module $\mathcal{M}$ its own signature $Sig_{\mathcal{M}}$ and variables $Var_{\mathcal{M}}$, its set of rules expressed with terms in $\mathcal{T}(Sig_{\mathcal{M}}, Var_{\mathcal{M}})$ and strategies. An ELAN program can be translated into a term in an extended signature $Sig_{\mathcal{B}}$ using the representation function. Let $Mod_{\mathcal{B}}$ be an ELAN module that defines terms $\mathcal{T}(Sig_{\mathcal{B}}, Var_{\mathcal{B}})$ built on this signature $Sig_{\mathcal{B}}$ (close to the ELAN syntax for programs) and variables $Var_{\mathcal{B}}$.

The module $Mod_{\mathcal{B}}$ implements the universal theory U for rewriting logic and the considered class of ELAN rewrite theories. $Mod_{\mathcal{B}}$ is actually composed of several modules, mainly a module `msTerm` introducing signatures and many-sorted terms, `rwrule` building rewrite rules with conditions and local affectations, `proofterm` defining proof terms that record subproofs. Instead of giving extensively the module $Mod_{\mathcal{B}}$, we would rather illustrate a part of it focussing on the representation of many-sorted terms, rewrite systems and proof terms. Meanwhile we also illustrate below several features of the ELAN language more detailed in [KKV95a,KKV95b,BKK+96b].

3.1 Representing many-sorted terms

¿From a list of identifiers `Types`, the module `type` (see Fig. 3) builds base sorts (`type`) and a more complex functional type (`sig`). Here, the quantification `FOR EACH...SUCH THAT` means that identifiers are extracted from the list `Types` and declared as constant operators of sort `type`. `@` is the same as `_` in the previous section.

This previous module `type`, combined with two lists of identifiers, is used by the module `msTerm` (see Fig. 4) to build operators and variables. More precisely, `msTerm` is parameterised by three lists: `Types` to import `type` with the correct parameter; `Vars`, a list of pairs of identifiers and types to define variables with their associated sorts; and `Ops`, a list of pairs of identifiers and signatures to define operators with their domains and codomains.

The line `F(@ {,@}~(N-1)) :({term}~N) term` in the module `msTerm` is expanded by the ELAN preprocessor into `F(@,...,@) : (term,...,term) term` according to the arity of `F`.

```
module type[Types]
sort type sig;
op global
        FOR EACH X:identifier SUCH THAT X:=(listExtract) elem(Types)
                :{ X : type; }
        @ -->> @    : (list[type] type) sig
        dom(@)      : (sig) list[type];
        cod(@)      : (sig) type;
endop
...
end of module
```

Fig. 3. Types built from identifiers

```
module msTerm[Vars,Ops,Types]
import  global msTermCommons
op global
        FOR EACH X:pair[identifier,type]
        SUCH THAT X:=(listExtract)elem(Vars)
                :{ X            : variable; }
        FOR EACH F:pair[identifier,sig]
        SUCH THAT F:=(listExtract) elem(Ops) ANDIF arity(F)==0
                :{ F            : term;}
        FOR EACH F:pair[identifier,sig]; N:int
        SUCH THAT F:=(listExtract) elem(Ops) AND N:=() arity(F)
        ANDIF N > 0
                :{ F(@ {,@}~(N-1)) :({term}~N) term; }
endop
...
end of module
```

Fig. 4. Many-sorted terms

3.2 Representing rewrite systems and proof terms

FIG. 5 shows how to represent rewrite rules in ELAN; types, signatures and many-sorted terms are described in `msTerm`. Rewrite systems are represented as lists of rewrite rules. FIG. 6 illustrates the construction of proof terms.

```
module rwrule[Vars,Ops,Types,Labels]
import  global msTermCommons
                label[Labels];
sort rwrule rulebody;
op global
        @ => @                  : (term term) rulebody;
        @ if @                  : (rulebody term) rulebody;
        @ where @ := @          : (rulebody variable term) rulebody;
        [] @ end                : (rulebody) rwrule;
        [@] @ end               : (label rulebody) rwrule;
endop
...
end of module
```

Fig. 5. Rewrite rules in ELAN

Based on the signature defined by $Mod_{\mathcal{B}}$, the representation function can be defined for terms, rewrite rules and theories and proof terms. As in [CM96], we use an overloaded function symbol $\overline{(\_)}$ to define the representation function.

- for R a set of rewrite rules $\{r_1, \ldots, r_n\}$, $\overline{R} = \overline{r_1}.\ldots.\overline{r_n}$;
  for an empty set R, $\overline{R} = \texttt{nil}$

```
module proofterm[Vars,Ops,Types,Rwrules,Labels]
import     global     list[proofterm];
           local      msTermCommons
                      msApplySubstOn[term]
                      label[Labels];
sort proofterm;
op
 global
           @;@                    : (proofterm proofterm) proofterm;
           @(@)                   : (label substitution) proofterm;
           @[[@]]                 : (proofterm list[proofterm]) proofterm;
endop
```

Fig. 6. Proof terms

- for r a rewrite rule $u \to u'$, $\overline{r} = \overline{u}$ `=>` $\overline{u'}$.
  for r a rewrite rule $u \to u'$ **if** c **where** σ, $\overline{r} = \overline{u}$ `=>` $\overline{u'}$ `if` $\overline{c}$ `where` $\overline{\sigma}$
- for t a term $f(t_1, \ldots, t_n)$, $f \in Sig_{\mathcal{M}}$, $n > 0$, $\overline{t}$ = `f(t_1,...,t_n)`
  where `f(@,...,@):(term ... term) term`,
  for t a term c (constant), $\overline{t}$ = `c` where `c : term`,
  for t a term x (variable), $\overline{t}$ = `x` where `x : variable`.
- for a proof term π,
  - if π is $\pi_1;\pi_2$, $\overline{\pi} = \overline{\pi_1};\overline{\pi_2}$
  - if π is $\ell(\sigma)$, $\overline{\pi} = \overline{\ell}(\overline{\sigma})$
  - if π is $\pi'\{\lambda\}$, $\overline{\pi} = \overline{\pi'}[[\overline{\lambda}]]$

With this translation, we can already represent simple programs (i.e. without strategies) by terms. It remains then to represent the entailment relation $\vdash$, which is the goal of the next Section 3.3.

Example 3.1 Let us consider the following program and suppose that we want to transform this meta-level object into its representation. In the signature defined by $Mod_{\mathcal{B}}$, the rewrite system becomes a term of sort `list[rwrule]`. Precisely the list

```
[] elem(cons(e,l)) => e end .
[] elem(cons(e,l)) => choice where choice:=elem(l) if non_empty_list(l)
end .  nil.
```

```
module list
import element
sort element list
op global
           nil                    : list;
           cons(@,@)              : ( element list ) list;
endop
rules for element
declare      e,choice : element;  l : list;
bodies
 [] elem(cons(e,l)) => e                  end
 [] elem(cons(e,l)) => choice
           where choice:=elem(l)
           if non_empty_list(l)
 end
end of rules
end of module
```

3.3 Simulating rewriting

In order now to encode rewriting in ELAN, it is needed to describe all the steps of matching, substituting and replacement in an elementary rewrite step and how to iterate them. Moreover it is actually required to encode conditional term rewriting with a special kind of *local affectation* inside the rules, introduced by the key word **where**. These local affectations are memorised during the elementary rewrite step in a substitution.

The rewriting mechanism is described with rewrite rules, applying on tuples $\pi : (t, \omega, r, \sigma, \mu)$ where π is a proof-term, t is the term to be rewritten, ω a position in t ($\wedge$ is used when no position is specified), r a rewrite rule in a set R of rewrite rules, σ and μ are substitutions. Note that sequents appearing in each rule are decorated with appropriate proof terms.

The representation function for sequents is defined as a rewrite step on a term built with a ternary operator $\_ \bullet \_ : \_$ taking as arguments a rewrite theory, a proof term and a term:

$$\overline{T \vdash \pi \ : \ t \to t'} = \overline{T} \bullet \ \overline{t} \ : \ \overline{t} \ \to \ \overline{T} \bullet \ (\overline{t}; \overline{\pi}) \ : \ \overline{t'}$$

We introduce a **one_step** rewrite rule, applying on such triple:

$$\textbf{one\_step}\ T \bullet t : t \quad \to \quad T \bullet (t; \pi) : t'$$

$$\text{where} \quad \pi, t', r \text{ such that } r \in R, \text{ and}$$

$$t : (t, \wedge, r, Id, Id) \to t; \pi : (t', \wedge, r, Id, Id)$$

This rule chooses a candidate rule r in R, performs an elementary step to rewrite the term t into t' and produces the associated proof term π. Let r be the candidate rule to rewrite the term t. Its left and right-hand sides are denoted respectively $lhs(r)$ and $rhs(r)$. There are four phases in the elementary rewrite step: find a position ω in t and a substitution σ, such that $\sigma(lhs(r)) = t_{|\omega}$, eliminate *conditional parts* by normalising conditions and compare them to $\top$, ($\top$ is the built-in boolean value *true* and is deeply connected to the implementation of conditions in rewrite rules), eliminate *local affectations* by recording them in substitution σ, and then apply the substitution and the replacement to obtain the resulting term $t[\sigma(rhs(r))]_\omega$. Before applying the replacement, phases 2 and 3 (elimination of *conditional parts* and *local affectations*) are iterated until no more condition or local affectation is left. We can express this algorithm with four rewrite rules **match**, **if_elim**, **where_elim** and **replace** detailed below and a simple strategy: (**match**; **if_elim***; **where_elim***; **replace**). The associated proof term is built in a deterministic way, during the computation. This provides a description of the proof closer to the actual execution in ELAN, since the proof term corresponds exactly now to the computation trace provided by the system.

The notation $proof(t \to t')$ is used to select the proof term in the sequent $\pi \ : \ \langle t \rangle \to \langle t' \rangle$. The function nf applied to a term t computes the normal form of t with respect to the whole set of rewrite rules R. Below we describe the rules that allow to compute an elementary rewrite step.

$$\textbf{match}\ \pi \ :\ (t, \wedge, r, Id, Id) \ \rightarrow\ \pi; (\ell(\sigma)\{\Box_{pt}\}) \ :\ (t, \omega, r, \sigma, \sigma)$$
$$\text{where } \omega, \sigma \text{ such that } \sigma(lhs(r)) = t_{|\omega}$$

Starting with a term t and a rewrite rule r, the **match** phase records a position ω in t and a substitution σ such that $\sigma(lhs(r)) = t_{|\omega}$. The rule first builds the proof term $\ell(\sigma)$ corresponding to the application of the rewrite rule r labelled by ℓ and instantiation of variables occuring in its left-hand side by values given by σ. $r(\sigma)\{\Box_{pt}\}$ is the proof term under construction. The list of subproofs issued from local affectations in this matching phase is initialised by the empty list denoted by $\Box_{pt}$.

$$\begin{aligned}\textbf{where\_elim}\ \pi \ :\ & (\pi'\{\lambda\}) : (t, \omega, r\ where\ v := t', \sigma, \mu) \ \rightarrow \\ & \pi; (\pi'\{\lambda.proof(\mu(t') \rightarrow t'')\}) \ :\ (t, \omega, r, \sigma \circ \{v \mapsto t''\}, \\ & \qquad\qquad \mu \circ \{v \mapsto t'\}) \\ & \text{where } t'' = nf(\mu(t'))\end{aligned}$$

This rule eliminates *one* local affectation. The rewrite rule r is transformed by removing the local affectation $v := t'$. The second substitution μ records this local affection while the first substitution σ records the normal form of t' w.r.t the rewrite system. The substitution σ will be useful to instantiate the right-hand side of r in the last **replace** phase.

Each time a local affectation is eliminated, μ records its initial form. So, instantiating a term by μ means eliminating new variables that do not occur in the left-hand side of r. To achieve the construction of the proof term associated to the application of r, the proof term associated to the sequent $\mu(t') \rightarrow t''$ (denoted by $proof(\mu(t') \rightarrow t'')$) is recorded in the list of subproof terms λ.

$$\begin{aligned}\textbf{if\_elim}\ \pi; (\pi'\{\lambda\}) \ :\ & (t, \omega, r\ if\ c, \sigma, \mu) \ \rightarrow \\ & \pi; (\pi'\{\lambda.proof(\mu(c) \rightarrow \top)\}) \ :\ (t, \omega, r, \sigma, \mu) \\ & \text{if } nf(\mu(t')) = \top\end{aligned}$$

This rule eliminates *one* condition. The rewrite rule r is transformed by removing the condition $if\ c$. The application of **if_elim** rule has only a side effect, it builds the subproof term associated to the normalisation of $\mu(c)$ into $\top$: $proof(\mu(c) \rightarrow \top)$.

$$\begin{aligned}\textbf{replace}\ \pi; (\pi'\{\lambda\}) \ :\ & (t, \omega, r, \sigma, \mu) \ \rightarrow \\ & \pi; t[\pi'\{\lambda\}]_\omega \ :\ (t[\sigma(rhs(r))]_\omega, \wedge, r, Id, Id)\end{aligned}$$

At this stage of the calculus, the substitution σ is able to instantiate each variable of the right-hand side of r by a ground term. The replacement is performed and produces the result term, as well as the resulting proof term. Several applications of **Congruence** are simulated by building the proof term

$t[\pi'\{\lambda\}]_\omega$ describing the *replacement* under the appropriate context t.

Soundness and completeness of this encoding is expressed in the two next propositions whose proofs are given in Appendix.

Proposition 3.2 *For any application of strategy (***match***;* **if_elim****;* **where_elim****;* **replace***, there exists a finite number of applications of rules* **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity** *building the same sequent and associated proof term.*

Proposition 3.3 *For any application of rules* **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity**, *there exists a finite number of applications of the strategy (***match***;* **if_elim****;* **where_elim****;* **replace***), building the same sequent and an* equivalent *associated proof term.*

Example 3.4 Let us consider different proofs of $g(f(a)) \to f(b)$ using the following rules: $r_1 \quad : \quad a \to b$, and $r_2 \quad : \quad g(x) \to x$. With rules **Reflexivity**, **Congruence**, **Replacement** and **Transitivity**, applied in different orders, one can derive three sequents with different proof terms:

$$\begin{aligned} r_2(f(r_1)) \quad &: \quad g(f(a)) \to f(b) \\ r_2(f(a)) \quad &: \quad g(f(a)) \to f(a) \\ \text{and } g(f(r_1)); r_2(f(b)) \quad &: \quad g(f(a)) \to g(f(b)) \end{aligned}$$

But only the two last ones can be derived using the strategy (**match**; **if_elim***; **where_elim***; **replace**). Indeed this is not a problem, since according to the Parallel Move Lemma on proof terms, the first one is equivalent to the two others.

These rewrite rules **match**, **if_elim**, **where_elim** and **replace** can easily be implemented in ELAN. We give below the strategy `rewrite_state` corresponding to (**match**; **if_elim***; **where_elim***; **replace**) and the ELAN program corresponding to an elementary rewriting step. The rules `if-elim` and `where-elim` are calling a strategy `rewrite` that corresponds to the normalisation process with all rules in R. Its implementation in ELAN is also given below.

```
strategy rewrite_state
  dont care choose(match)
  repeat
    dont care choose(if_elim where_elim)
  endrepeat
  dont care choose(replace)
end of strategy
```

```
rules for rewrite_state
declare
        s,t,g,d,c,a             : term;
        v                       : variable;
        rw                      : rwrule;
        rb                      : rulebody;
        sigma                   : substitution;
        omega                   : list[int];
        mc                      : contrainte;
        lab                     : label;
        pi,pi2                  : proofterm;
        lpi                     : list[proofterm];
        couple                  : pair[term,proofterm];

bodies
 [match] [s,nil,lab,rb,identity]:pi =>
        [s,omega,lab,rb,sigma]:pi;lab(sigma)[[nil]]
                where sigma := () contrainte_to_subst(mc)
                where mc := (matchs) g =? s at omega
                where omega := (chooseOccurence) nvocc(s)
                where g := () left(rb)
 end

 [if_elim] [s,omega,lab,rb if c,sigma]:pi;pi2[[lpi]] =>
        [s,omega,lab,rb,sigma]:pi;pi2[[second(couple).lpi]]
                if first(couple)==true
                where couple:=(rewrite) [sigma(c),proofnil]
 end

 [where_elim] [s,omega,lab,rb where v:=a,sigma]:pi;pi2[[lpi]] =>
        [s,omega,lab,rb,sigma o v->t]:pi;pi2[[second(couple).lpi]]
                where t:=() first(couple)
                where couple:=(rewrite) [sigma(a),proofnil]
 end

 [replace] [s,omega,lab,rb,sigma]:pi;pi2[[lpi]] =>
        [s[sigma(d)] at omega,omega,lab,rb,identity]:pi;s[ pi2[[lpi]] ] at sigma
                where d := () right(rb)
                if no_conditional_no_local_affectation_rewrite_rule(rb)
 end
end of rules
```

```
rule rewrite for pair[term,proofterm]
declare
        s,t             : term;
        rs              : rewrite_state;
        rwr             : rwrule;
        pi,pi2          : proofterm;
body
 [s,pi] => [t,pi;pi2]
        where pi2:=() rewrite_state_to_proofterm(rs)
        where t :=() rewrite_state_to_term(rs)
        where rs :=(rewrite_state) rewrite(s,rwr)
        where rwr:= (listExtract) elem(rwrules)
end of rule

strategy rewrite
  repeat
    dont care choose(rewrite)
  endrepeat
end of strategy
```

4 Implementation

In order to provide now a suitable implementation of the representation function, we propose a translation in two steps. We distinguish between the representation of meta-level objects (described in $Mod_{\mathcal{B}}$) and a low-level translation mechanism, which should be implemented independently from $Mod_{\mathcal{B}}$. So the user can now choose his/her own representation of terms or rewrite rules in a flexible manner. More formally, we define the representation function as the composition $\varphi_2 \circ \varphi_1$ where $\varphi_1 : \mathcal{T}(Sig_{\mathcal{M}}, Var_{\mathcal{M}}) \mapsto \mathcal{T}(Sig_{\mathcal{B}'}, Var_{\mathcal{B}'})$ and $\varphi_2 : \mathcal{T}(Sig_{\mathcal{B}'}, Var_{\mathcal{B}'}) \mapsto \mathcal{T}(Sig_{\mathcal{B}}, Var_{\mathcal{B}})$, where $Sig_{\mathcal{B}'} \subseteq Sig_{\mathcal{B}}$ and $Var_{\mathcal{B}'} \subseteq Var_{\mathcal{B}}$. φ_1 encodes the translation mechanism that transforms any meta-level object into a fixed low-level representation (here we choose a list of identifiers as example), and φ_2 corresponds to the transformation of this low-level representation into user defined representation described by $Mod_{\mathcal{B}}$. Fig. 7 illustrates this idea.

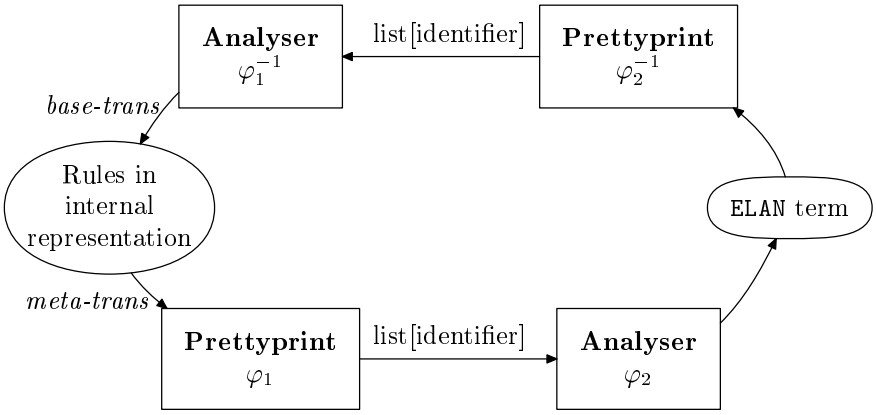

Fig. 7. Representation mechanism

The low-level representation consists only in identifiers, a binary concatenation operator '.' and a constant operator `nil` to terminate the list. Identifiers are built from strings of ASCII characters. In the implementation, it is needed to distinguish identifiers from operators '.' and `nil`, which compels us to introduce quotes '"'.

Instead of giving extensively the specification of φ_2 and φ_2^{-1}, we illustrate on an example what they do. Then we present two considered approaches to implement them.

Example 4.1 Given a term $t \in \mathcal{T}(Sig_{\mathcal{M}}, Var_{\mathcal{M}})$ such that $t = f(a, b)$. Its low-level representation is a term of $\mathcal{T}(Sig_{\mathcal{B}'}, Var_{\mathcal{B}'})$, namely the list of identifiers `op.{.f.}.[.a.b.].nil` (or more precisely `"op"."{"."f"."}"."["...`). An application of φ_2 on this representation transforms it into a term $t' \in \mathcal{T}(Sig_{\mathcal{B}}, Var_{\mathcal{B}})$: φ_2(`op.{.f.}.[.a.b.].nil`) = `f(a,b)`. φ_2^{-1} does the converse: φ_2^{-1}(`f(a,b)`) = `op.{.f.}.[.a.b.].nil`.

The first investigated approach consists in specifying φ_2 (resp. φ_2^{-1}) with rewrite rules. This amounts to describe a syntax analyser with rewrite rules that depend on the term representation described in $Mod_{\mathcal{B}}$. The following

piece of code illustrates an implementation of φ_2 for the representation adopted in Section 3, FIG. 4.

```
module phi2[Vars,Ops,Types]
op global
        phi2(@)                 : (list[identifier]) term;
        revphi2(@)              : (term) list[identifier];
endop
rules for term
declare
        F : identifier;
        L : list[identifier];
bodies
 []     phi2( "op" . "{" . F . "}" . "[" @ L @ "]" . nil ) =>
        F( FOR EACH X:identifier SUCH THAT X:=(listExtract) elem(L)
                :{ phi2(X), } )
 end
 []     phi2( "op" . "{" . F . "}" . "[" . "--" . "]" . nil ) => F
 end
end of rules

rules for list[identifier]
declare
        T : term;
bodies
 []     revphi2(T) =>
        "op" . "{" . head(T) . "}" . "[" .
                { revphi2(I-th subterm(T)) .}_I=1...arity(head(T)) "]" . nil
        if arity(head(T)) > 0
 end
 []     revphi2(T) =>
        "op" . "{" . head(T) . "}" . "[" . "--" . "]" . nil
        if arity(head(T)) == 0
 end
end of rules
```

Another solution to implement φ_2 would be to use the Earley algorithm implemented in ELAN to parse the low-level representation. Adopting this solution means that the low-level representation is no more accessible via `list[identifier]` and becomes an intermediate data structure immediately translated into a term of $\mathcal{T}(Sig_{\mathcal{B}}, Var_{\mathcal{B}})$ by a built-in function calling the Earley algorithm.

This presentation clearly shows the two built-ins φ_1 and φ_1^{-1} that have to be added to get a flexible reflective programming environment. φ_1 cannot be implemented with rewrite rules, because it has to access to information stored in C++ structures corresponding to meta-level objects. This mechanism has to be implemented in C++ and to be called as a *built-in* in ELAN programs. We think φ_1 is not really difficult to implement, since it is just a function that reads information in the memory. Conversely, φ_1^{-1} modifies the memory state and the ELAN programs. Implementing this functionality seems the most technical and difficult task to achieve in order to get interesting reflective capabilities.

5 Conclusion

In the literature, work on reflection covers a lot of different approaches.

Since in particular the seminal works of Gödel, Turing and Feferman, lo-

gicians have been mainly concerned with meta-reasoning, i.e. the ability to analyse inference steps or proofs in a given object logic using a second layer of logic called meta-logic. A meta-theory can be used to describe an object theory, to derive statements about the object theory, to control the search or to talk about provability in the object theory. Reflection up and down are two inference rules mentioned in [GT92] to formalise switching levels, and to theoretically justify for instance the combination of object-level and meta-level proofs and search in theorem proving. Logical frameworks and theorem provers, such as the systems FOL [Wey80,GS89] and GETFOL [Giu92,GT92], NuPRL [C+86,KC86], or ELF [Pfe94] to cite a few, are providing interesting meta-reasoning capabilities. A survey and critique on metatheory and reflection in theorem proving can be found in [Har95].

On the other hand, in computer science, reflection takes its roots in universal Turing machines and functions. Designers of programming languages have been most interested, as we are in this paper, by achieving, thanks to reflection, an extensible and flexible computation mechanism, and the ability to read and modify sources of programs. Several reflective languages have been designed in functional programming [Smi84], logic programming [BK82], object-oriented programming [Mae87] and rewrite system based languages [KSO95].

There is some hope of providing metalogical foundations for reflection in [CM96] that would unify many different approaches, and this is the reason why we adopted this view in our work. Complementary aspects of reflection in ELAN are mentioned in [BKK96a].

Acknowledgements

We sincerely thank Claude Kirchner and Peter Borovanský for helpful discussions and comments.

References

[BK82] K. Bowen and R. Kowalski. Amalgamating language and metalanguage in logic programming. In *Logic programming, APIC Studies in Data Processing*, volume 16, pages 153–172. Academic Press, 1982.

[BKK96a] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling rewriting by rewriting. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[BKK+96b] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4, Asilomar (California), September 1996. Electronic Notes in Theoretical Computer Science.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1986.

[CM96] M. G. Clavel and J. Meseguer. Axiomatizing Reflective Logics and Languages. Proceedings of Reflection'96, pages 263–288. Xerox PARC, 1996.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Giu92] F. Giunchiglia. The GETFOL manual-GETFOL version1. Technical Report 9204-01, IRST, Trento, Italy, 1992.

[GS89] F. Giunchiglia and A. Smail. Reflection in constructive and non-constructive automated reasoning. In H. Abramson and M. Rogers, editors, *Meta-Programming in Logic Programming*, pages 123–140. MIT Press, 1989.

[GT92] F. Giunchiglia and P. Traverso. A metatheory of a mechanized object theory. Technical Report 9211-24, IRST, Trento, Italy, 1992.

[Har95] J. Harrison. Metatheory and reflection in theorem proving: A survey and critique, 1995.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal of Computing*, 15(4):1155–1194, 1986. Preliminary version in Proceedings 11th ACM Symposium on Principles of Programming Languages, Salt Lake City (USA), 1984.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[KC86] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proceedings, Symposium on Logic in Computer Science*, pages 237–248, Cambridge, Massachusetts, 16–18 June 1986. IEEE Computer Society.

[KKV95a] C. Kirchner, H. Kirchner, and M. Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, pages 131–158. The MIT press, 1995.

[KKV95b] C. Kirchner, H. Kirchner, and M. Vittek. ELAN *V 1.17 User Manual.* Inria Lorraine & Crin, Nancy (France), first edition, November 1995.

[KM95] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proceedings 6th Conference on Rewriting Techniques and Applications, Kaiserslautern (Germany)*, volume 914 of *Lecture Notes in Computer Science*, pages 438–443. Springer-Verlag, 1995.

[KSO95] M. Kurihara, T. Sato, and A. Ohuchi. Reflective computation in term rewriting systems. *Computer Software*, 12(4):3–14, 1995.

[Mae87] P. Maes. Concepts and experiments in computational reflection. In N. Meyrowitz, editor, *Proceedings of OOPSLA'87*, ACM, pages 147–155. ACM, 1987. Special issue of SIGPLAN Notices, Volume 22, Number 4.

[Mes89] J. Meseguer. General logics. In H.-D. E. et al., editor, *Logic Colloquium'87*, pages 275–329. Elsevier Science Publishers B. V. (North-Holland), 1989.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[MOM93] N. Marti-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. Technical report, SRI International, 1993.

[Pfe94] F. Pfenning. Elf: A meta-language for deductive systems (system description). In A. Bundy, editor, *12th International Conference on Automated Deduction*, LNAI 814, pages 811–815, Nancy, France, June 26–July 1, 1994. Springer-Verlag.

[Smi84] P. C. Smith. Reflection and semantics in lisp. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programmings Languages*, ACM, pages 23–35. ACM, 1984.

[Vit94] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, October 1994.

[Wey80] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, (13):133–170, 1980.

6 Appendix

To improve readability, we denote by **sim** the strategy (**match**; **if_elim***; **where_elim***; **replace**). We also write $\pi \ : \ t \to t'$ instead of $\pi \ : \ \langle t \rangle \to \langle t' \rangle$.

Proposition 3.2.

For any application of the strategy **sim**, *there exists a finite number of applications of rules* **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity** *building the same sequent and associated proof term.*

Let r be a rewrite rule labelled by ℓ:

$\ell \ : \ u(\overline{x}) \to u'(\overline{x} \cup \overline{y})$
if $v_1(\overline{x} \cup \overline{y}) \to v'_1(\overline{x} \cup \overline{y}) \wedge \ldots \wedge v_j(\overline{x} \cup \overline{y}) \to v'_j(\overline{x} \cup \overline{y})$
where $\sigma = \{y_1 \mapsto a_1(\overline{x}), \ldots, y_k \mapsto a_k(\overline{x} \cup \overline{y}^{k-1})\}$

Without loss of generality, we can formulate r as:

$\ell \ : \ u(\overline{x}) \to u'(\overline{x} \cup \overline{y})$
if $v_1(\overline{x} \cup \overline{y}) \to \top \wedge \ldots \wedge v_j(\overline{x} \cup \overline{y}) \to \top$
where $\sigma = \{y_1 \mapsto a_1(\overline{x}), \ldots, y_k \mapsto a_k(\overline{x} \cup \overline{y}^{k-1})\}$

Let us prove Proposition 3.2 by induction on the depth n of the proof term resulting from the application of the strategy **sim**. The *depth* of a proof term is defined by:

- $\mathrm{depth}(\pi_1;\pi_2) = \max(\mathrm{depth}(\pi_1), \mathrm{depth}(\pi_2))$
- $\mathrm{depth}(t) = 0$
- $\mathrm{depth}(f(\pi_1,\ldots,\pi_n)) = \max_{i=1,\ldots,n}(\mathrm{depth}(\pi_i))$
- $\mathrm{depth}(\ell(\overline{\alpha})\{\beta_1 \ldots \beta_k.\gamma_1 \ldots \gamma_j\}) =$
  $1 + \max(\max_{i=1,\ldots,k}(\mathrm{depth}(\beta_i)), \max_{i=1,\ldots,j}(\mathrm{depth}(\gamma_i)))$

▶ **case** $n = 1$**:** Then there is no condition or local affectation in r, $(j + k = 0)$. Applying rules **match** and **replace** on the tuple: $t : (t, \wedge, u(\overline{x}) \to u'(\overline{x}), Id, Id)$ yields:

$t : (t, \wedge, u(\overline{x}) \to u'(\overline{x}), Id, Id)$

$$\xrightarrow{\textbf{match}} t; \ell(\sigma)\{\Box_{pt}\} : (t, \omega, u(\overline{x}) \to u'(\overline{x}), \sigma, \sigma)$$
$$\xrightarrow{\textbf{replace}} t[\ell(\sigma)\{\Box_{pt}\}]_\omega : (t[\sigma(u'(\overline{x}))]_\omega, \wedge, u(\overline{x}) \to u'(\overline{x}), Id, Id)$$

Ground terms appearing in σ and their associated proofs terms $\overline{\alpha}$ are build by **Reflexivity**; an application of **Replacement** builds the sequent: $\ell(\overline{\alpha}) \ : \ t_{|\omega} \to u'(\{\overline{x} \mapsto \overline{w}\})$. Several applications of **Congruence** transform the sequent $\ell(\overline{\alpha}) \ : \ t_{|\omega} \to u'(\{\overline{x} \mapsto \overline{w}\})$ into $t[\ell(\overline{\alpha})]_\omega \ : \ t \to t[u'(\{\overline{x} \mapsto \overline{w}\})]_\omega$. The composition of **match** and **replace** corresponds to an application of the **Transitivity** deduction rule.

▶ **case** $n + 1$**:** Then there are n embedded applications of one-step rewriting in the evaluation of conditions and local affectations in r. Let us assume that we apply the rules **match**, m times $(m = j + k \geq 1)$ **where_elim** or **if_elim** and then **replace** on the tuple: $t : (t, \wedge, r, Id, Id)$ and get the result:

$$t; t[\ell(\sigma)\{\lambda\}]_\omega : (t[u'(\{\overline{x} \cup \overline{y} \mapsto \overline{a} \cup \overline{w}\})]_\omega, \wedge, u(\overline{x}) \to u'(\overline{x} \cup \overline{y}), Id, Id)$$

where $\lambda = \beta_1 \ldots \beta_k.\gamma_1 \ldots \gamma_j$ is a list of proof terms β_i and γ_i corresponding to evaluation of conditions and local affectations in r and of depth at most n.

By induction hypothesis applied to the proof terms in λ and the associated reductions, there is a finite number of applications of rules of rewriting logic that build these proof terms and sequents. An application of **Replacement** followed by several applications of **Congruence** then build the resulting sequent and proof term.

Proposition 3.3.

For any application of rules **Reflexivity**, **Congruence**, **Replacement** *and* **Transitivity**, *there exists a finite number of applications of the strategy* **sim** *building the same sequent and an* equivalent *associated proof term.*

Let prove by structural induction on proof term the expected result.
Suppose that sequents involved in premises of rules **Reflexivity**, **Transitivity**, **Congruence** and **Replacement** can be obtained in an *equivalent* form by applying **sim**. We want to prove that for each sequent produced by one application of those deduction rules, there exists a given sequence of applications of **sim** that produce an equivalent sequent.

▶ **Reflexivity:** This is trivially obtained by 0 application of the strategy **sim**.

▶ **Transitivity:** Let $\pi_1 \quad : \quad t_1 \to t_2$ and $\pi_2 \quad : \quad t_2 \to t_3$ be two sequents. By hypothesis, there exist a finite number of applications of **sim** that transforms the term t_1 (resp. t_2) into t_2 (resp. t_3) and produces the proof term π_1' (resp. π_2') equivalent to π_1 (resp. π_2). Successive applications of those two sequences of **sim** produce the proof term: $\pi_1'; \pi_2'$ which is equivalent to $\pi_1; \pi_2$.

▶ **Congruence:** Given $\pi_i \quad : \quad t_i \to t_i', \ i = 1 \dots n$. An application of **Congruence** produces the sequent

$$f(\pi_1, \dots, \pi_n) \quad : \quad f(t_1, \dots, t_n) \to f(t_1', \dots, t_n')$$

By induction hypothesis, equivalent sequents $\pi_i' \quad : \quad t_i \to t_i', \ i = 1 \dots n$ can be produced by **sim**.

Starting from the term $f(t_1, \dots, t_n)$, successive applications of **sim** produce:

$$f(\pi_1', t_2 \dots, t_n); f(t_1', \pi_2', t_3 \dots, t_n); \dots; \ f(t_1', \dots, t_{n-1}', \pi_n') \to \\ f(t_1, \dots, t_n) \quad : \quad f(t_1', \dots, t_n')$$

which is a proof term equivalent to $f(\pi_1', \dots, \pi_n')$ so also to $f(\pi_1, \dots, \pi_n)$. This equivalence uses repeatedly axioms of Independence and Local Identities.

▶ **Replacement:** Given $\alpha_i, i = 1 \dots n$, $\beta_i, i = 1 \dots k$ and $\gamma_i, i = 1 \dots j$. By induction hypothesis, the list λ of subproof terms is equivalent to the list $\overline{\beta}^k.\overline{\gamma}^j$. An application of the (**match**, ..., **replace**) sequence at the top position ϵ produces the proof term $t[\ell(\sigma)\{\lambda\}]_\epsilon = \ell(\sigma)\{\lambda\}$ which is equivalent to $\ell(\overline{w}^n)\{\overline{\beta}^k.\overline{\gamma}^j\}$. Successive applications of **sim** will produce:

$$\ell(\overline{w}^n)\{\lambda\}; u'(\alpha_1', w_2, \dots, w_n); u'(w_1', \alpha_2', w_3, \dots, w_n); \dots u'(w_1', \dots, w_{n-1}', \alpha_n')$$

which is a proof term equivalent to $\ell(w_1, \dots, w_n)\{\lambda\}; u'(\alpha_1', \dots, \alpha_n')$ (thanks to Independence and Local Identities of FIG. 2).

Applying the axiom Parallel Move Lemma of FIG. 2, this proof term is equivalent to $\ell(\alpha_1', \dots, \alpha_n')\{\lambda\}$ where the $\alpha_i', i = 1 \dots n$ are equivalent to $\alpha_i, i = 1 \dots n$. The resulting proof term is equivalent to $\ell(\overline{\alpha}^n)\{\overline{\beta}^k.\overline{\gamma}^j\}$.